6.841 Advanced Complexity Theory Feb 20, 2007

Lecture 4

Lecturer: Madhu Sudan *Scribe: Katherine Lai*

1 Overview

In this lecture, we look at some lower bounds concerning the non-uniform forms of computation we discussed last time.

- Neciporuk's Theorem ($f \in P$ with BP size $\Omega((\frac{n}{\log n})^2)$)
- Barrington Construction of Small-Width BPs

2 Previous Lecture

In the previous lecture, we discussed non-uniform forms of computation and their complexity measures.

Nonuniform computation	# bits of advice	time	size	width	depth
TM with advice	x	x			
Circuits			x		x
Branching Programs (BPs)			x	x	
Formulae			x		x

We try to study these forms of computation to try to see what you can and can't solve with them, and in doing so divide up logspace and other classes.

3 Counting Technique

Before we discuss Neciporuk's theorem, we define the counting technique, which we will find useful.

Technique 1 *Let* $\mathcal{F} \subseteq \{f : \{0,1\}^n \to \{0,1\}\}$. *Then* $\exists f \in \mathcal{F}$ *s.t.* $size(f) \geq \Omega\left(\frac{log|\mathcal{F}|}{\log\log|\mathcal{F}|}\right)$.

We can use this technique to count the number of functions circuits of a particular size can calculate. If we have circuits with size at most s, they can calculate $|\mathcal{F}| \leq 2^{s \log s}$.

However, if we wanted to use only this technique to prove anything meaningful about NP, it's never going to really work. NP is an infinite family of functions. For this concept to make sense, we would need to look at the first constant number of functions in NP, and we don't get anywhere if we use counting to find one function in NP. We will, however, see how we can get lower bounds with counting anyway.

4 Neciporuk's Theorem

With Neciporuk's Theorem, we get a lower bound on the size of BP that calculates a function in P. First, we define the function.

Definition 2 *Let* $f : \{0,1\}^k \times \{0,1\}^{n-k} \to \{0,1\}$*, with* $f_x(i) = f(i,x)$*.*

The question at hand is, how many different f_x do we get? If all f_x's are different, can we say anything about $size(f)$? We want to find out what the smallest circuit or BP we would need for this f. The key idea is if we have a circuit for f, we have a circuit for f_x–we just hardcode x into the circuit. This doesn't increase the size of the circuit. If all f_x are different, we have

$$\mathcal{F} = \{f_x\} \text{ with } |\mathcal{F}| \geq 2^{n-k}$$

$$size(f) \geq \frac{n-k}{\log(n-k)} = \Omega\left(\frac{n}{\log n}\right)$$

This result is unfortunately sublinear, so it doesn't tell us much yet.

To construct $\mathcal{F}$ with a different f_x for every x, we do the following. We set $f(i,x) = x_i$, where x is the truth table where $x = x_1 \dots x_{n-k}$, provided that $2^k \geq n-k$.

4.1 A superlinear lower bound

We want to find a superlinear lower bound, and to do this we have to restrict ourselves to only using BPs. We also need a metric for the size of the BP.

Definition 3 *Let* $BPSIZE_i(B)$ *be the number of times the literal* x_i *or its complement appears in the branching program* B*.*

In the case of branching programs, we actually have a big advantage. Take subset $S \subseteq 1 \dots n$ $BPSIZE_S(B) = \#$ edges labelled by literals of S in the branching program B. To get a BP for f_x from one for f, we simply fix the literals and shortcircuit some of the edges and erase other ones.

If f is a function on n variables and $S_1, S_2 \dots S_l$ is a partition of the set $\{1 \dots n\}$, then $BPSIZE(f) \geq \sum_{i=1}^{l} BPSIZE_{S_i}(f)$

Definition 4 *Let the function Distinct? be defined as*

$$Distinct?(x_{11} \dots x_{1k}, x_{21} \dots x_{2k}, \dots, x_{l1} \dots x_{lk}) = \begin{cases} 1 \text{ if } \forall j \neq k, i_j \neq i_k \\ 0 \text{ otherwise} \end{cases}$$

with $n = lk$*.*

We would like to try to relate the lower bound to the number of bits. Take any block of variables. If we restrict the remaining variables, we get many different functions. There are $2^{k^{(l-1)}}$ ways to choose them all different for every

set $T \subseteq \{1 \dots 2^k\}$ with $T = i_2 < i_3 < i_4 \dots i_l$. From T, we can create the function

$$f_T(i_1) = \begin{cases} 1 \text{ if } i_1 \notin T -> f_1(i_1) = f(i1, i2, ...il) \\ 0 \text{ if } i_1 \in T \end{cases}$$

The number of functions on S_1 is at least

$$\begin{pmatrix} 2^k \\ l-1 \end{pmatrix} \approx \begin{pmatrix} 2^k \\ l \end{pmatrix} \geq \left(\frac{2^k}{l}\right)^l$$

This is actually true for any S_i, so we get, by picking $k \geq 2 \log l$,

$$\begin{aligned} BPSIZE_{S_i}(f) &\geq \frac{\log(2^k/l)^l}{\log\log(2^k/l)^l} \\ &\geq \frac{kl - l\log l}{\log(kl)} \\ &\geq \Omega\left(\frac{n}{\log n}\right) \end{aligned}$$

Using the result from before, we have a total size

$$\begin{aligned} BPSIZE(f) &\geq \sum_{i=1}^{l} BPSIZE_{S_i}(f) \\ &\geq l \cdot \frac{n}{\log n} = \Omega\left(\frac{n}{\log n}\frac{n}{\log n}\right) \end{aligned}$$

and we achieve the desired superlinear lower bound. This may come in useful when we try to separate NP from NL or other complexity classes.

In the next lecture, we'll see different examples of restricting other values.

5 Barrington Construction of Small width BPs

5.1 Background

Roughly at stage of Neciporuk's Theorem, people were trying to separate NP from P or NL. Some thought that maybe looking at not just size, but size and width simultaneously, they would get some result. They hoped the following would be true.

Proposition 5 *If $f \in P$ needs large width branching programs (super polynomial), then $P \neq L$.*

The problem with this, however, is whether we can prove super polynomial large widths. Given any DNF formula, it is trivial to construct a BP for it with only three branches. For example, if we have the DNF formula $x_1x_2x_3 \vee \bar{x}_1x_3x_4 \vee x_2\bar{x}_3x_5$, the BP can sequentially try every term and branch between two of the branches, exiting to the third branch if it is clear that the whole formula will be satisfied already. Since any formula for $f \in P$ can reduce to a DNF formula, no f needs super polynomial width. This leads to the next idea.

Proposition 6 *If $\exists f \in P$ that needs large branching programs– super polynomial width or super polynomial size then $P \neq L$.*

This turns out to be true, but we still don't know how to prove super polynomial size. We will instead pursue a weaker goal: we will try to find $f \in P$ that requires super constant width or super polynomial size. (This doesn't actually prove anything.)

Proposition 7 *The function*

$$Majority(x_1 \dots x_n) = \begin{cases} 1 \text{ if } \sum x_i \geq n/2 \\ 0 \text{ otherwise} \end{cases}$$

requires super constant width if the size is polynomial.

Theorem 8 *Barrington's Theorem: Let f be a function that has a depth d formula over {binary AND, NOT}. Then f has a BP of width 5 (we will prove 8 today) and depth 4^d. It has a log-depth formula, so it has a polynomial-sized BP.*

5.2 Register Machine

To prove Barrington's theorem, we will be using a register machine [Ben-Or,Cleve]. A register machine consists of l registers, $R_1, \dots, R_l$, each of which stores one bit. The register machine is fed a set of instructions $I_1, \dots, I_S$ that describe how to update the registers. For example, I_2 might be $R_1 \leftarrow R_2 + x_i \cdot R_3$. The end result is then some linear combination of the registers.

If a register machine takes $(R_1 \dots R_l) \rightarrow (R_1 \dots R_{(l-1)}, R_l + f(x_1 \dots x_n) \cdot R_1)$, then this machine is said to compute f. This can be done by setting the registers to $(1, 0, \dots, 0)$ initially. This leaves the first $l-1$ registers the same and the answer in R_l.

Fact 9 *If an l-register machine of size S computes f, then $\exists BP$ of width 2^l and size $O(S)$.*

The width comes from needing to remember all registers, of which must have at least 2^l states. To prove the BP of width 8, we will thus need a 3-register machine.

5.3 Ben-Or Cleve Induction

If f has depth d formula over {binary NOT, AND}, then f is computed in size 4^d by a 3-register machine.

To calculate NOT, or $\neg f_1$ for some f_1, we use the following register machine. The instructions are:

$$\begin{array}{ll} I_1: & R_3 \leftarrow R_3 + f_1 R_1 \\ I_2: & R_3 \leftarrow R_3 + R_1 \end{array}$$

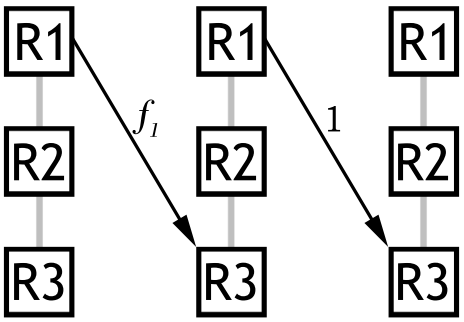

Figure 1: Diagram of a BP used for a NOT operation

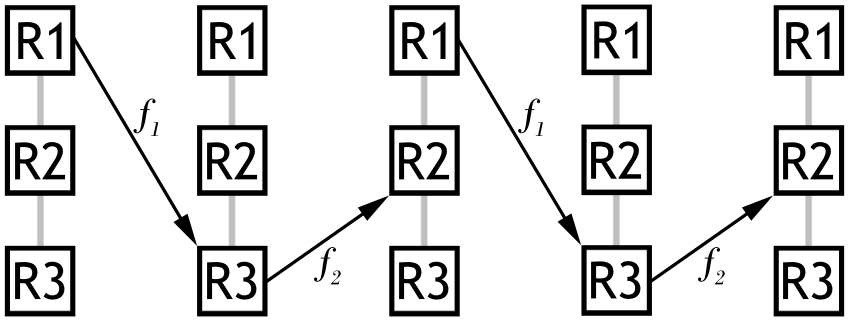

Figure 2: Diagram of a BP used for an AND operation

which leaves the first two registers unchanged, and changes R_3 to contain the value $r_3 + (1 + f_1)r_1$, where r_1 and r_3 are the initial values in registers R_1 and R_3 respectively. We can then obtain the value $\neg f_1$ from the result in R_3.

To calculate AND between f_1 and f_2, it is somewhat more complicated. We use the following instructions for the register machine:

$$\begin{array}{ll} I_1\text{:} & R_3 \leftarrow R_3 + f_1 R_1 \\ I_2\text{:} & R_2 \leftarrow R_2 + f_2(R_3 + f_1 \cdot R_1) \\ I_3\text{:} & R_3 \leftarrow R_3 + f_1 R_1 \\ I_4\text{:} & R_2 \leftarrow R_2 + f_2 R_3 \end{array}$$

If we let the initial values in the registers be r_1, r_2, and r_3, we see that I_3 had the effect of restoring R_3 to its original value r_3, and I_4 cancelled out the $f_2 r_3$ term, so that in the end, R_2 contained $r_2 + f_1 f_2 r_1$.

5.4 Conclusions

At the end of Barrington's results, there were many objects coming together and looking identical–log-depth circuits, formulas, polynomial size and bounded width BPs, and so on. Counting has not worked to show if circuits and formulae are different. In actuality, the conjecture mentioned earlier is false. We know functions that require super polynomial size, but they're not in PSPACE, and the question is how far away that is. Relatedly, the complexity class "Nick's class" NC^i is for circuits of depth $\log^i n$, and it is named after Nick Pippenger.